CMPSC-265
Data Structures and Algorithms

Zaihan Yang
zyang13@suffolk.edu

Department of Math and Computer Science
Suffolk University

Fall 2019

Notice

- Midterm_Exam 1 grade posted.
- HW8 posted. Will be due on this Sunday midnight (11.59pm)
- Computer Science Open House Activity:
  - Time: Oct 29th, Tuesday
  - Location:
  - You are Welcome to attend!

Recap

- Binary Tree
  - Basic concepts and terms
  - Coding practice: how to get the height of a binary tree.
  - How to determine whether two binary trees are identical to each other
  - Traversal algorithms:
    - Breadth-first traversal (level-order traversal)
    - Depth-first traversal:
      - Pre-order (root-left-right)

Binary Tree

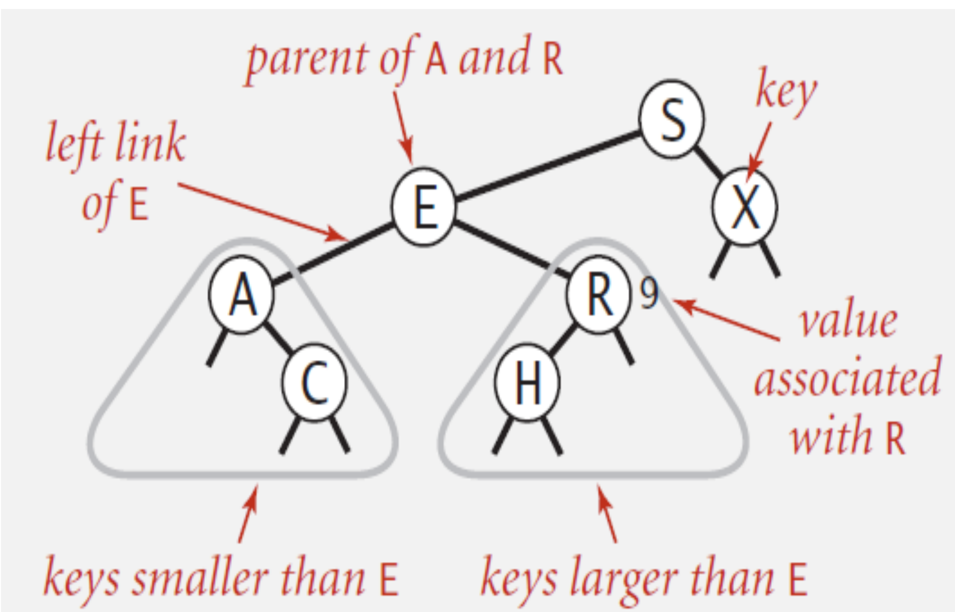
- Depth-first Traversal:
  - Pre-Order: root – left – right
  - In-Order: left – root – right
  - Post-Order: left – right – root

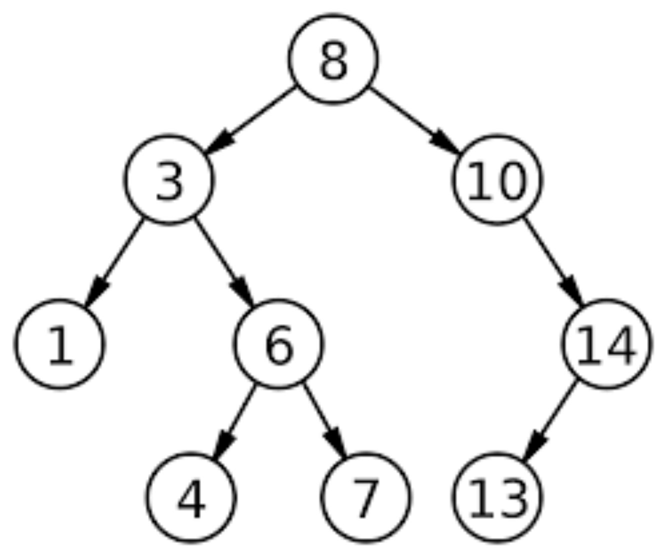
Time Complexity for Traversal

Traversal	Time Complexity
Pre-Order	O(n)
In-Order	O(n)
Post-order	O(n)

Binary Search Tree(BST)

- It arranges data (keys) in a way to make search efficient.
- It is a binary tree in which for every node we have both of the following conditions:
  - Left subtree contains keys less than its root node
  - Right subtree contains keys greater than or equal to its root node





BST Class

```
class Tree {
        private Node root;// Reference to root

        // constructor
        public Tree() {
                root = null;
        }

        public Node find(int key) {
                //…
        }
        public void insert(int key) {
                //…
        }
        public Node delete(int key) {
                //…
        }
}
```

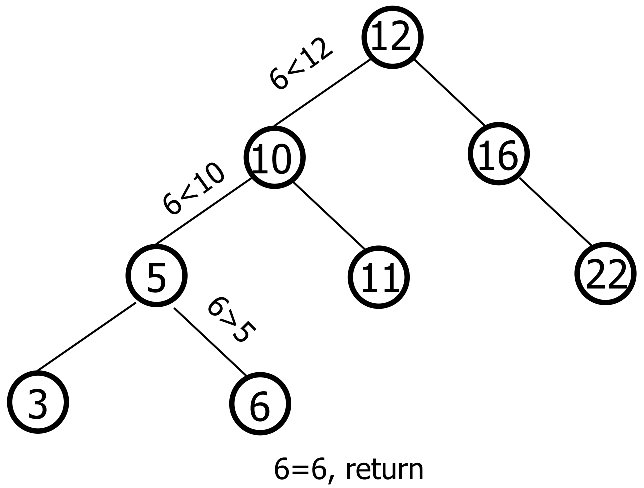
```
class Node {
        public int key; // key value
        public Node left; // left child
        public Node right;// right chilc

        // constructor
        public Node(int key) {
        this.key = key;
        left = null;
        right = null;
        }
}
```

Binary Search Tree Basic Operations

- Find a node
- Insert a node
- Find minimum in a BST
- Delete a node
- Traverse the BST (the same as Binary Tree)

Finding a Node in a BST

- Take advantage of the BST structure/property, compare the search key with root and decide which subtree to continue
- Example, find(6)



Finding a Node in a BST

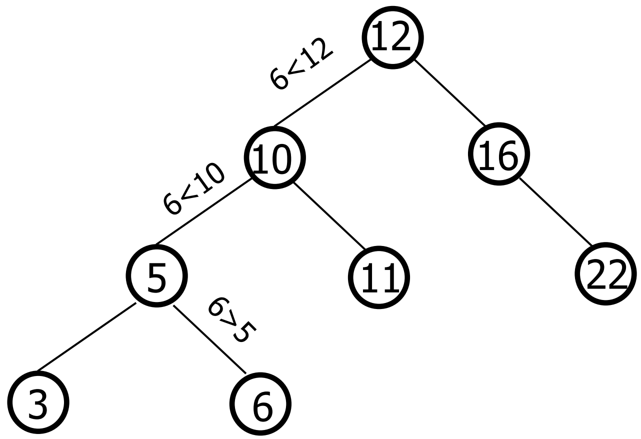
- Recursive

```
public Node find(Node current, int target) {
        if (current==null || target==current.key)
                return current;

        if (target < current.key) // continue on the left side
                return find(current.left, target);
        else // continue on the right side
                return find(current.right, target);
}
```

- What is the time complexity?
  - O(h) where h is the height of the tree. If balanced h=logn

Inserting a Node in a BST

- We have to search (similar to find()) to find the right place for insertion.
- If empty, create a new node and insert as root
- Example, insert(6)



Inserting a Node in a BST

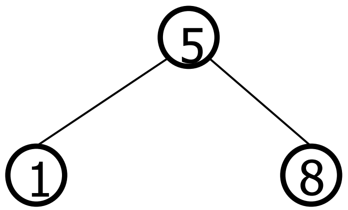
```
public void insert(int key) {
        root = recInsert(root, key);
    }

public Node recInsert(Node root, int key) {

        // If empty, create a new node and return it
        if (root == null) {
            root = new Node(key);
            return root;
        }

        // otherwise search left or right
        if (key < root.key)
            root.left = recInsert(root.left, key);
        else if (key >= root.key)
            root.right = recInsert(root.right, key);

       return root;
    }
```

Traversing a Binary Tree

- The same of the traversal algorithm on general binary tree
- We want to visit every node (once).
- Recursive approach makes it conceptually easy.
- We have different traversal methods depending on the order of visit
  - Pre-order
  - In-order
  - Post-order

In-order Traversal

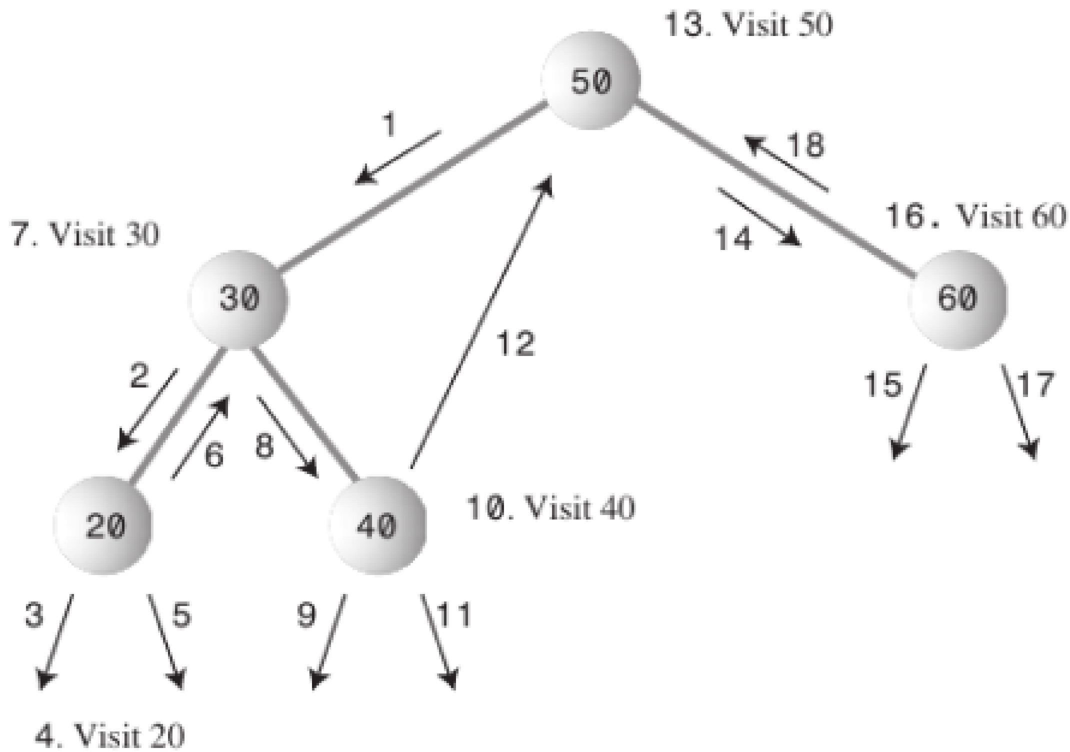
- Here is the algorithm:
  - Visit left
  - Visit the node
  - Visit right



1 5 8

```
public void inOrder(Node root)
{
        if (root!=null){
                inOrder(root.left); //visit left
                System.out.print(root.key + ” ”);//visit root
                inOrder(root.right); //visit right
        }
}
```

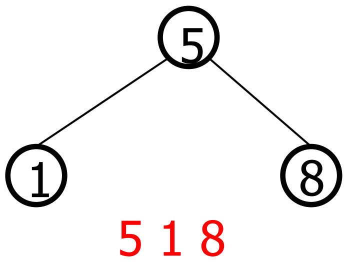
In-order Traversal



- In-order traversal output: 20, 30, 40, 50, 60
- In-order traverse of the BST will result in an ascending sequence of all the node values

Pre-order Traversal

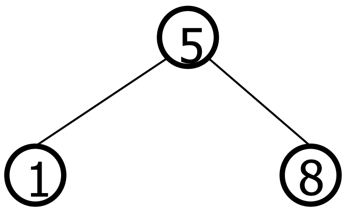
- Here is the algorithm:
  - Visit the node
  - Visit left
  - Visit right




```
public void preOrder(Node root)
{
        if (root!=null){
                System.out.print(root.key + ” ”);//visit root
                preOrder(root.left); //visit left
                preOrder(root.right); //visit right
        }
}
```

Post-order Traversal

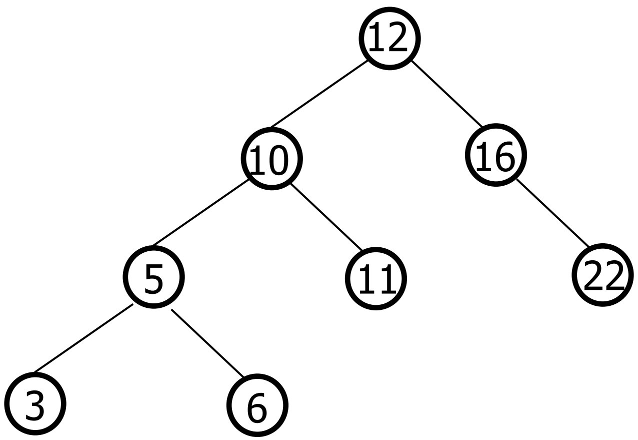
- Here is the algorithm:
  - Visit left
  - Visit right
  - Visit the node



1 8 5

```
public void postOrder(Node root)
{
        if (root!=null){
                postOrder(root.left); //visit left
                postOrder(root.right); //visit right
                System.out.print(root.key + ” ”);//visit root
        }
}
```

Example- Binary Tree Traversal

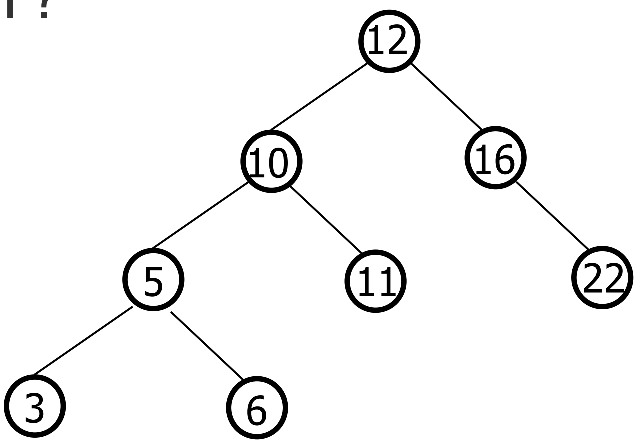


- In-order: 3 5 6 10 11 12 16 22
- Pre-order: 12 10 5 3 6 11 16 22
- Post-order: 3 6 5 11 10 22 16 12

- Is this a BST? What can we say about in-order traversal of a BST?
  - yes, in-order traversal of a BST results in a sorted sequence.
- What is the time complexity of binary tree traversal?
  - O(n)

Find Minimum in a BST

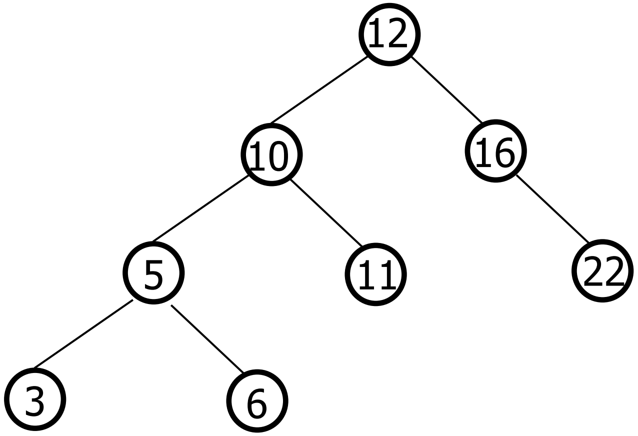
- Where to find the minimum in a BST?
  - The left most node




```
public Node min()
{
        Node current,last;
        current=root;
        while(current!=null){ //until the bottom
                last=current; //remember node
                current=current.left; // go left
        }
        return last;
}
```

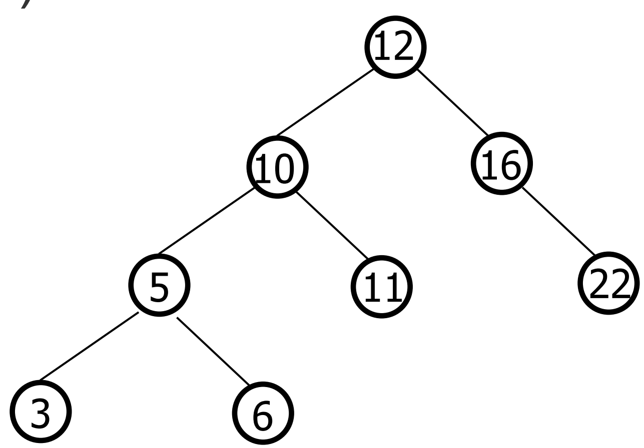
Deleting a Node in a BST

- We need to keep the BST structure/property
  - 3 possible cases
    - Node is a leaf
    - Node has one child
    - Node has two children



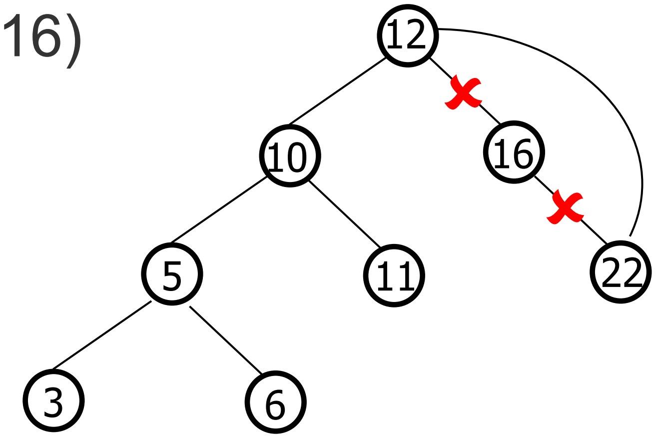
Deleting a Node in a BST - Case 1

- The node which is a leaf needs to be disconnected from its parent.
- First, we need to find the node, remember its parent and remember if it was a right child or left child of its parent.
- Then, set the relevant child reference of parent to null.
- Example: delete(6)



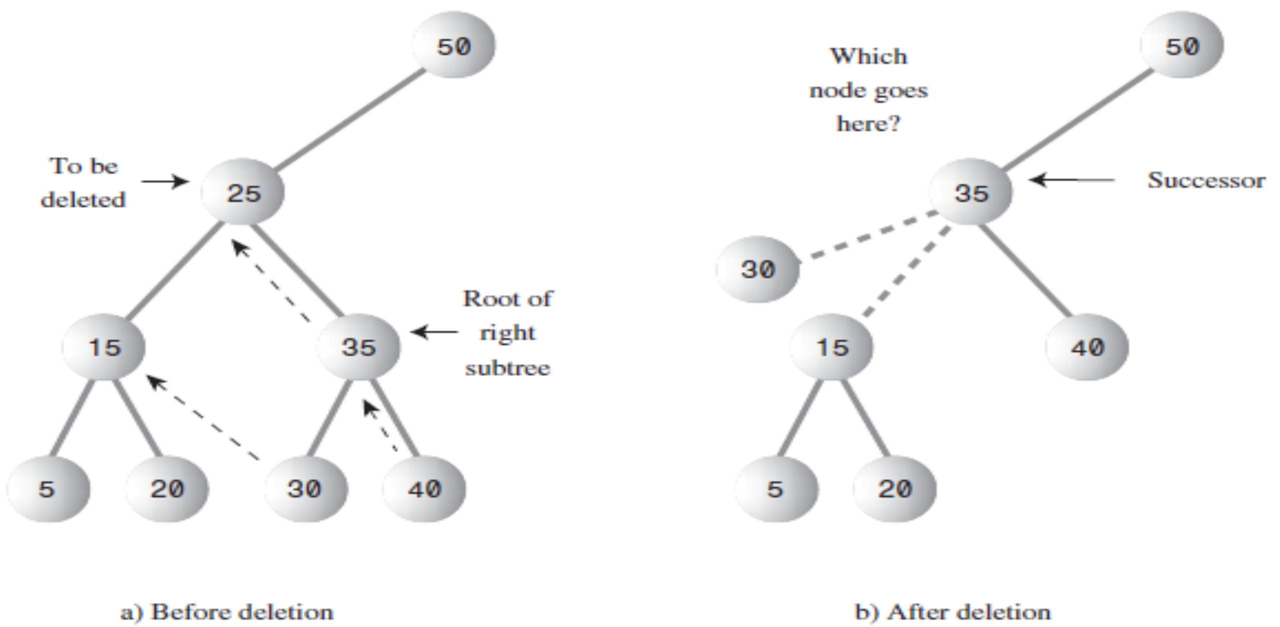
Deleting a Node in a BST - Case 2

- The node has one child.
- First, we need to find the node, remember its parent, remember if it was a right child or left child of its parent, and remember if its child is on left or right.
- Then, set the relevant child reference of parent to its only child.
- Example: delete(16)



Deleting a Node in a BST - Case 3

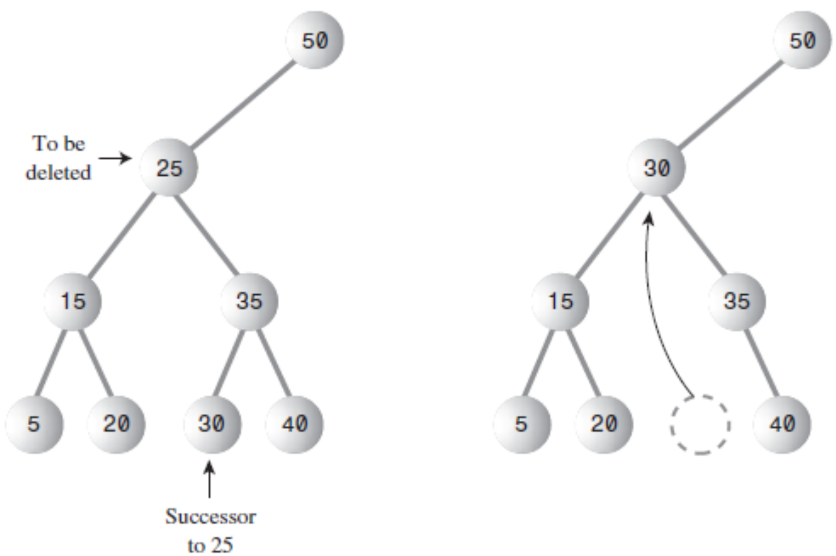
- The node has two children.
- We cannot simply lift a subtree
- Example: delete(25)



Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 394

Deleting a Node in a BST - Case 3

- We replace the node with its (*in-order) successor*
  - In-order successor of node n is the minimum on the right subtree of n, or the node which comes after n if we traverse the tree in-order



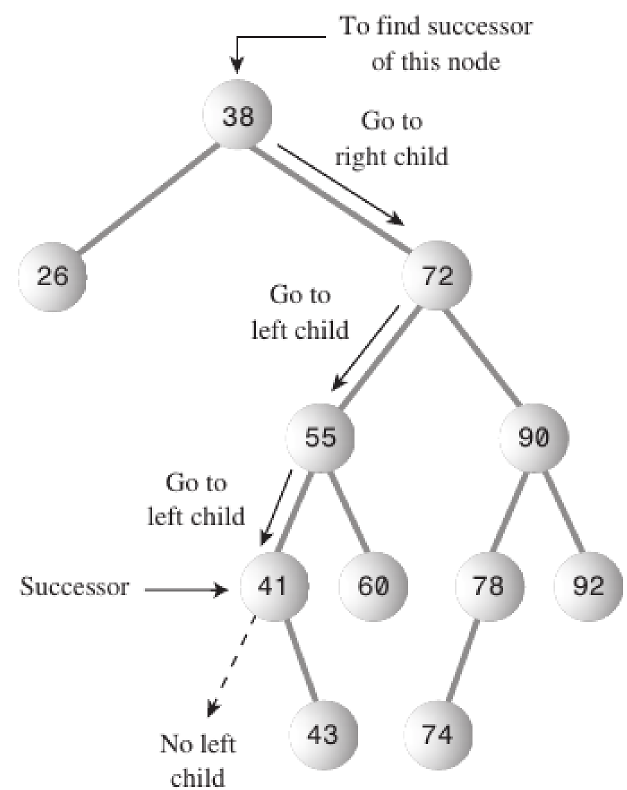
a) Before deletion b) After deletion

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 394

Finding the Successor of a Node

- Minimum in the right subtree
  - Note that successor does not have a left child

```
public Node getSuccessor(Node delNode)
{
  Node successorParent = delNode;
  Node successor = delNode;
  Node current = delNode.right;

  while(current!=null){
    successorParent=successor;
    successor=current;
    current=current.left;
  }

  if (successor != delNode.right){
    successorParent.left=successor.right;
    successor.right=delnode.right;
  }

  return successor;
}
```



Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 395